Lab1

Perera N N D
210467A

Part1

Approach:

1. **Command Parsing:**

   The input command string was parsed using `strtok()` with `+` as the delimiter.Each token (command) was stripped of leading and trailing spaces. An empty command case was handled by setting default arguments for `execve()`.

2. **Command Execution:**

   Each parsed command was executed in a child process created by `fork()`. The child process used `execve()` to replace its image with the command to be executed. The parent process waited for the child process to complete using `waitpid()`.

Challenges Faced:

1. **Handling `strtok()` and Command Separation:**

   After using `strtok(input, "+")` to get the first command, I needed to handle subsequent commands. The challenge was that `strtok()` with `NULL` would not work directly due to its usage in the `execute_command()`function.

   To address this, I placed a `\0` character after each command and used `end + 3` to skip the empty space between commands, which allowed continued parsing. This approach avoided issues with overlapping `\0` characters but required careful management to ensure commands were parsed correctly.

2. **Avoiding Duplicate Execution:**
   - Proper handling of the command parsing and process creation was necessary to avoid executing the same command multiple times inadvertently. Checking if a process is a child process using pid help me to solve this.

Part 2

Approach:

1. **Command Parsing and Execution:**

   The program reads the command and placeholders from the command line arguments. Placeholders are counted to determine if dynamic input is required. If placeholders are present (`%`), the program reads lines of input from stdin and replaces placeholders with these inputs. Each command is executed with the provided arguments. Commands are executed using `execvp()` in a child process created by `fork()`. The parent process waits for the child to complete before continuing.

2. **Memory Management:**

   The program uses `strdup()` to allocate memory for command arguments and placeholders. Proper memory management is crucial to avoid segmentation faults. After command execution, dynamically allocated memory is freed to prevent memory leaks.

Challenges Faced:

1. **Memory Management Issues:**

   Improper memory management led to segmentation faults. This was primarily due to incorrect handling of dynamically allocated memory and not freeing allocated pointers appropriately.

   Had to ensure that all dynamically allocated memory was freed after use. This included freeing memory allocated by `strdup()` and ensuring that all pointers were properly managed and updated.

2. **Handling Placeholders and Dynamic Input:**

   Replacing placeholders with dynamic input required careful management of argument arrays. The code had to ensure that placeholders were correctly replaced and that the arguments were properly terminated with `\0`. Implemented logic to handle dynamic input by updating argument arrays based on the number of tokens read from stdin. After processing each line of input, the arguments were reset to handle subsequent commands correctly.